




Containerizing the PowerAPI Architecture to Estimate Energy Consumption of Software Applications

Daniel Guamán¹^a, Alejandra Barco-Blanca², Vanessa Rodríguez-Horcajo²^b and Jennifer Pérez²^c

¹Universidad Técnica Particular de Loja, Loja, Ecuador

²Universidad Politécnica de Madrid, Madrid, Spain

Keywords: Software Engineering, PowerAPI, RAPL, Energy Consumption, Sustainability, Software Containerization.

Abstract: The widespread adoption of cloud architectures and the use of information technologies have a significant impact on software sustainability, particularly in terms of energy consumption. PowerAPI is a toolkit designed to estimate the energy consumption of software applications. It integrates hardware performance counters (HWPC) and SmartWatts formulas to analyze energy usage at different abstraction levels, providing enough accurate estimation metrics to drive an energy-efficient software design. However, its configuration deployment may be complex. In this work, we aim to extend its use by facilitating its deployment. To that end, we present a study that explores the containerization of PowerAPI in two different measurement contexts. From the results of this study, a middleware solution to estimate the energy consumption of software applications, called *PowerAPIDocker-Cloud*, has been constructed. *PowerAPIDocker-Cloud* implements a scalable and reproducible energy consumption monitoring process in two different contexts: (i) Java Model-View-Controller (MVC) desktop monolithic applications and (ii) containerized microservices MVC applications written in different programming languages. The experimentation carried out during the study demonstrates the feasible measurement of 29 applications in the first context and 4 applications in the second context. The set of experiments show that *PowerAPIDocker-Cloud* is a reusable mechanism to easily and effectively estimate the energy consumption of MVC software applications using PowerAPI. In addition, the experiments contribute insights into how to design energy-efficient architectures and to identify resource-efficient programming techniques that can contribute to reduce the environmental impact of MVC software applications in containerized environments.

1 INTRODUCTION


The adoption of cloud architectures and the widespread use of information technologies play a key role in the sustainability of software, influencing its development and use (Kocak, 2013). Industries, organizations, and individuals using cloud software and services contribute significantly to energy consumption, posing challenges in software sustainability. Energy efficiency, as a key aspect of sustainable development, seeks to optimize energy consumption throughout the entire lifecycle of software, from design to implementation, making requirements specification a critical factor in improving software energy performance (Agarwal et al., 2012).


In the context of energy efficiency, Green IT and

Green Software establish practices, frameworks and metrics to promote sustainability in the development and evolution of software products (Bozzelli et al., 2013). These approaches seek to reduce software complexity by optimizing its quality, performance and energy consumption at different levels of abstraction, such as architecture, source code, instructions and deployment platforms, both on-premise and in the cloud (Pazowski et al., 2015).

Research on monitoring, evaluation and management of energy consumption in software and infrastructure has led to the definition of sustainability principles applicable to software engineering and architecture, as well as energy metrics and measurement tools, such as jRAPL, (Liu et al., 2015), RAPL (David et al., 2010) and PowerAPI (Fieni et al., 2024). Evaluating the energy efficiency of software is a key aspect in sustainability, since operating costs are considered to mitigate the environmental impact and re-

^a <https://orcid.org/0000-0002-2681-565X>

^b <https://orcid.org/0009-0007-6401-6078>

^c <https://orcid.org/0000-0003-3192-7995>

duce the carbon footprint of software products. However, despite the advances in Green Software and software sustainability, a challenge within the field is to measure, analyze and optimize energy consumption through standardized tools that could be used in different contexts. Setting up and using these measurement tools is not a simple task. Therefore, in this work, *PowerAPIDocker-Cloud* is presented as a containerized middleware to effectively collect the energy data in different environments using PowerAPI and allow software engineers to make informed decisions for dealing with energy-efficient software design. Configuring and deploying PowerAPI in a containerized environment addresses the need for a scalable and reproducible energy consumption monitoring infrastructure independently of the kind of software under measurement. To evaluate the effectiveness of *PowerAPIDocker-Cloud* and its versatility, it has been validated in two different contexts: (i) monolithic desktop Java applications built with the MVC architectural pattern, and (ii) containerized MVC applications built with microservices and programmed in different languages (Torvekar and Pravin, 2019).

This paper is organized as follows: Section 2 describes the required background about PowerAPI and its toolkit. Section 3 presents the construction of *PowerAPIDocker-Cloud*. Section 4 details the experimentation and evaluation of *PowerAPIDocker-Cloud*. Section 5 discusses the results obtained from the study execution. Finally, the conclusions and future work are presented in Section 6.

2 PowerAPI

Grant et al. (Grant et al., 2016) describe *PowerAPI* as a middleware able to measure energy consumption at different levels: system components, software process and user usage, providing a complete set of energy metrics. PowerAPI's power consumption estimation relies on the configuration of user-defined sensors that collect raw data and process it using a *Power Model*, i.e. an energy consumption model that enables power consumption estimation in software systems (Fieni et al., 2024).

Figure 1 shows the software power consumption estimation process that PowerAPI implements. This process consists of 5 steps: **(1)** A *sensor* collects raw metrics from the computer where the software applications under energy consumption monitoring are being executed, being incompatible its use in a virtual machine. **(2)** *Metrics* (*power*, *cpu usage*) are stored in a database to be used by the software power consumption estimation model. **(3)** The *Power Estimation*

Model uses machine learning techniques to estimate the power consumption of software applications using the collected raw data measurements. This model is auto-calibrated when it is necessary. **(4)** The *energy consumption estimated values* that are generated by the Power Estimation Model are stored in another database. **(5)** Finally, these values are used to optimize the software applications under measurement from an energy consumption standpoint.

The software power meter of PowerAPI is composed of two components: the *HWPC sensor* and the *formulas* (see Figure 1). **(a)** The *HWPC sensor* generates detailed low-level metrics that can be controlled directly from the processor based on the *RAPL* technology. The *RAPL interface formula* obtains the consumption of the entire CPU, the information from the system registers (MSR) about the energy consumption and the *SmartWatts formula* processes the data of each system process individually. The *SmartWatts* formula defines a power estimation model based on a linear regression model of the *scikit-learn* library (Pedregosa et al., 2011), which is self-calibrated by using appropriate performance counters and an error threshold provided by the power meter.

3 PowerAPIDocker-Cloud: A CONTAINERIZED SOLUTION FOR MEASURING ENERGY CONSUMPTION USING PowerAPI

3.1 Deploying PowerAPI in a Containerized Environment

PowerAPIDocker-Cloud implements the *PowerAPI* architecture in a containerized environment (see Figure 1). The containerization is carried out through the installation and configuration of the sensor and formula, the integration of each component and the configuration of the platforms *Docker* (Rad et al., 2017), *Docker Compose* and *Docker Engine*. This containerized environment allows for collecting metrics of software applications energy consumption, independently of their development technology. In order to validate its feasibility, in this work we have deployed it in two different contexts that allow to measure MVC applications: **(i) Context 1:** Monolithic applications, and **(ii) Context 2:** Containerized microservices applications.

The goal of *PowerAPIDocker-Cloud* is to be a reproducible deployment model, independently of the

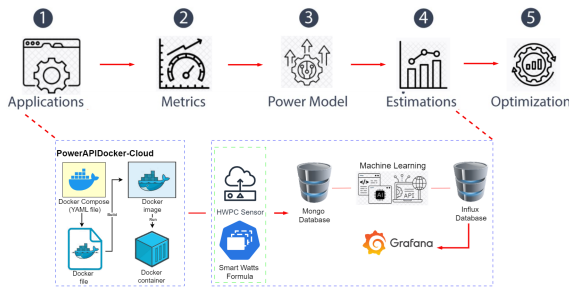


Figure 1: PowerAPIDocker-Cloud Architecture.

kind of software under measurement, which serves as a guide and support for researchers who require collecting energy consumption measurements. In both contexts, *PowerAPIDocker-Cloud* requires the cloud images of the Mongo and Influx Databases for storing the data and estimations, as well as the common sensor and the configurations. However, the effective containerization of *PowerAPI* in each context presents several differences and technical considerations that are detailed as follows:

Context 1: The MVC applications under measurement are programmed in Java. For their power consumption measurement, *PowerAPI* and its components are installed in *PowerAPIDocker-Cloud* in a *Docker Container*. To run each application under measurement, a NetBeans image is installed into the *Docker Container* of *PowerAPIDocker-Cloud*. In this way, the installation will allow consistency, reuse and portability of the *Docker Container*.

Context 2: The MVC applications are programmed with different programming languages implementing a microservices software architecture. In this context, unlike the previous scenario, several *Docker Containers* are required in *PowerAPIDocker-Cloud*, since each application has its own configurations. The capabilities offered by *Docker* allow for rapid code delivery, testing, and deployment, thus significantly reducing the time between code creation and deployment.

3.2 PowerAPIDocker-Cloud Configuration

PowerAPIDocker-Cloud is a guidance for software engineers and architectures about how to use *PowerAPI* in a containerized way in order to be reusable and replicable in different settings. In particular, this work addresses two different contexts of deployment. To that end, a set of common configurations must be performed. They are detailed as follows.

PowerAPI- HWPC Sensor: There are different options for using the HWPC sensor, which also uses the RAPL technology for monitoring the power con-

sumption of the CPU or RAM. In this work, since we seek to monitor the power consumption of the container of the application under measurement, the *CORE option* is selected.

PowerAPI-SmartWatts Formula: This formula receives the metrics provided by the HWPC sensor to estimate the software's energy consumption. Each time the CPU error threshold is reached (cpu-error-threshold), it will learn a new *Power Model* from previous reports. It is important to determine the CPU frequency of the hardware used for the measurement, since the formula uses this information to calculate energy consumption. The configuration for estimating the power consumption uses `RAPL_ENERGY_PKG` of RAPL (Running Average Power) and the TSC, APERF and MPERF events from the MSR (Model Specific Register) as components that allow access to a processor's energy meters.

Mongo and Influx Database: In these two contexts, *MongoDB* and *InfluxDB* are used to store the values of the measurements, that are processed by the *Power Model* through the *PowerAPI Machine Learning* process and then visualized by the *Grafana's Power Report*. The selected configuration uses *MongoDB* for storing the measurements, whereas *InfluxDB* is used for storing the energy consumption estimations.

Grafana: The measurement data stored in *Influx Database*, which are the energy estimation data calculated by the *SmartWatts* formula, are displayed in *Grafana* and then its dashboard is configured for presenting the energy consumption data in real time.

Dashboard to Visualize Data: To visualize energy monitoring data, a query must be configured in *Grafana* by selecting the data source, in this case, *InfluxDB* previously configured. In our case, for real-time monitoring, a display range of the last 10 minutes is configured, and the dashboard is updated every 5 seconds, allowing for dynamic and continuous monitoring.

4 EXPERIMENTATION AND EVALUATION OF PowerAPIDocker-Cloud

The experimental study conducted in this work to evaluate the containerized *PowerAPI* environment (*PowerAPIDocker-Cloud*) has a twofold objective: (i) to prove that it is feasible to measure and estimate the power consumption of applications in both contexts and (ii) to use these estimations to extract knowledge about the energy consumption behaviors of the kind of applications under measurement. In particu-

lar, three research questions were formulated to address these two objectives. RQ1 answers the first objective, whereas RQ2 and RQ3 answer the second one:

- RQ1: Is it possible to measure the energy consumption of software applications programmed with different software architectures and programming languages using a containerized PowerAPI environment?
- RQ2: Does the execution time of a functionality influence in the energy consumption or CPU values measured by PowerAPIDocker-Cloud?
- RQ3: Does the type of CRUD (Create-Read-Update-Delete) operation of an application influence in the energy consumption? What kind of operation (Create-Read-Update-Delete) has the highest energy consumption?

To rigorously address the experimental study, an execution context was established and a set of experiments were designed to measure the energy consumption of the software applications in both contexts.

To suitably measure the energy consumption of the applications using the PowerAPIDocker-Cloud in the two defined contexts, the configuration of the environment and the definition of the parameters used for the sensor and formula were defined. The activities of this process are detailed following: (1) Create the configuration file, (2) Start and run containers. In the first context, the NetBeans 8.2 IDE must be deployed, whereas in the second context, the container images and configurations for each application will be downloaded from GitHub. (3) Store the initial data generated by the measuring equipment prior to the measurement. (4) Start and run Grafana, (5) Execute the application to be monitored from the container that holds the application to be measured. The application is executed, and its modules and functionalities begin to be used for a specific period of time, ensuring that all the main functionalities are tested at least once and all applications are executed during the same time (Mancebo et al., 2021). (6) Measure the application, when the *Docker* containers and the *HWPC Sensor* services are started, the databases and the formula for obtaining energy consumption data in each container are also initiated. However, data measurement is saved in the *Influx Database* once the application is running. (7) With the data saved in Influx Database in real time, graphs are displayed in Grafana showing these data. To do this, we introduce the query that shows the last two minutes of the container's execution, while the output is exported in .CSV format from Grafana with the consumption data of each application with an interval of 1 second. It should be

noted that, although we can modify parameters in the dashboard display, it will be the query that responds to the exported file, for later analysis and comparison. These steps of the process are executed as many times as applications we need to measure.

4.1 Measurements in Context 1

The Context 1 is characterized for measuring a dataset of 29 desktop MVC applications implemented in Java. This dataset is obtained from the previous studies of Guaman et al. (Guamán et al., 2023)(Guamán et al., 2022). The measurement procedure and the results obtained are detailed as follows.

Measurement: The 29 characterized applications were measured in this experimentation. To determine the required execution time to evaluate all applications for the same duration without bias, we selected the ones with the most functionalities. This allows us to define a time period that covers the testing of all their features. From this execution we determine that 2 minutes was enough to execute all the functionalities at least once. Therefore, each of the 29 applications were measured during a period of 2 minutes by executing the query "*query time > now - 2m*". The energy consumption data is collected in an interval of 1 second each data, obtaining the energy consumption in Watts per second (W/s) during the 2 minutes of execution. The average energy consumption in W/s is detailed in Table 1.

Table 1: Java MVC applications monitored with PowerAPI energy consumption metrics.

AppID	AppName	Power Consumption (W/s)	Max-Min	Zeros	Standard deviation
App1	Biblioteca	1.51125097	7.17086	9	1
App2	BoulderDash	1.35492206	32.13968	1	5.13798
App3	SistemaContable	1.84270151	11.09284	17	1.57741
App4	deberExamen	1.62192057	4.75086	30	0.97357
App5	java-swing-contact	2.16258772	13.4	13	1.47648
App6	AlarmClock	1.68863052	7.00118	32	0.97936
App7	DesktopMVC	1.11631539	4.55808	40	0.95901
App8	CRUDMVC	1.4719136	4.55368	30	0.88716
App9	GestionBiblioteca1	1.43896392	11.93922	50	2.14367
App10	BrickBreaker	4.46842435	16.8118	12	1.98293
App11	java-swing-mvc-master	2.04786631	9.9404	7	1.66206
App12	java-tetris-mvc-master	0.98839078	5.61335	7	0.90928
App13	Agenda Electronica	8.50535088	14.78	0	3.03707
App14	NotaAlumnos	1.89566667	5.002	8	0.90711
App15	OperacionesBasicas	1.37094257	6.74822	2	1.3262
App16	VolumenFigurasTridimensionalesVista	1.62203684	8.38804	9	1.37443
App17	Registros	2.29467257	44.4	14	4.34381
App18	ConversorMonedesGUI	1.52991228	4.44	8	1.04188
App19	jc_mvc_demo	1.24958542	3.62785	27	1.03047
App20	07_SimpleDrawMVCVisitor-master	11.51008772	27.12	0	6.62809
App21	Ideal Weight Calculator	2.57842105	15.4	19	1.52872
App22	Matricula	1.76481579	22	53	4.58302
App23	EnsayoProgAvanzada1Bim	3.43081617	9.08962	11	2.00915
App24	EnsayoUtp1	6.80361404	83.3	14	11.32255
App25	HugoSacaquirin	3.16388597	21.368	0	3.46482
App26	CRUDPAQUETES	3.90933333	32.602	0	4.85256
App27	AlexanderReyes (PUCO)	1.95806053	5.1852	1	1.09165
App28	SwingMvc	0.84643562	23.45451	0	2.46201
App29	Boletos	1.02537668	5.48092	14	1.14343

Results and Analysis of Experiments: The 29 applications were effectively measured during the 2

minutes of execution using PowerAPIDocker-Cloud to search for power consumption behavioral patterns that could help software engineers better understand the power consumption behavior of software MVC applications. From this analysis, the first finding that we identified is that the power consumption measurements of the 29 applications reveal high energy consumption peaks when the application is started. This shows that the launch of NetBeans 8.2 within the container increases the energy consumption of the container. In addition to the power consumption average, Table 1 presents the standard deviation of each application.

A high standard deviation indicates greater variability in the data, which could suggest that there are consumption peaks or significant fluctuations in the application; whereas, a low standard deviation indicates less variability and, therefore, a more balanced and regular consumption. Based on the standard deviation results, we have classified the applications into 3 groups of applications: **(a) Balanced Group** ($x < 1.5$): those applications that have a standard deviation below 1.5 W/s. **(b) Moderate Variable Group** ($1.5 < x < 3.5$): those applications that have a standard deviation between 1.5 W/s and 3.5 W/s. **(c) High Variable Group** ($x > 3.5$): those applications that show a standard deviation above 3.5 W/s.

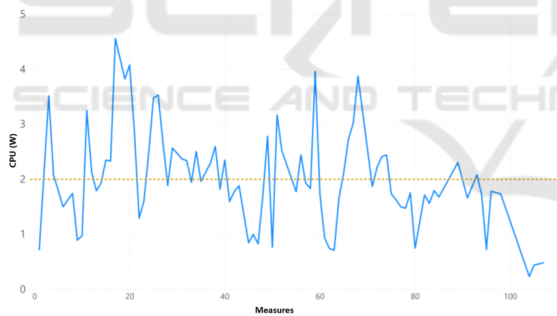


Figure 2: Example Group A: Balanced App8.

As a result, 14 of the 29 applications analyzed belong to *Group a: Balanced*, since the measurement values are close to the average and their consumption patterns show that they consumption behaviors are *relatively stable*. Figure 1 shows the energy consumption data of the *App8 CRUDMVC*, where at first glance it seems like an application with various fluctuations. However, the results vary between 0.5 W/s and 4.5 W/s, and most of these measurements are close to the total average of the application (2 W/s), so it follows a *stable and balanced consumption flow* (see Figure 2).

On the other hand, 9 applications are categorized within *Group b: Moderate Variable*, since the energy consumption values of each application range

between 1.5 W/s and 3.5 W/s, around the average of each application. Finally, it has been determined that 6 applications are characterized in *Group c: Variable high* due to the very disparate records in their execution exhibiting a typical deviation above 3.5 W/s in their energy consumption. This is shown in Figure 3, which shows fluctuations throughout the monitoring, varying up to 27 W/s from maximum to minimum consumption. In this *Group c*, there is also evidence of high consumption in the applications, since the average of 7.13 W/s exceeds the average of all the groups by more than 3.69 W/s of the total set. Therefore, *applications, that comply with a High Variability flow, present an energy consumption significantly higher than the average energy consumption of the applications of the rest of the patterns and the average of the total set.*

This is mainly because two of the most power-consuming applications belong to this group, *App20* - 11.51008772 W/s and *App2* - 13.66912519 W/s. Furthermore, the fourth application that consumes the most energy is also found in this group *App24* - 7.75612 W/s, with *App22* and *App17* being the only ones in this subset with energy consumption values below the total consumption average, although they are ranked eighth and eleventh in the order of those that consume the most.

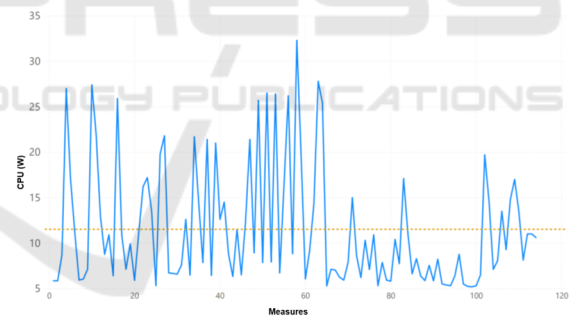


Figure 3: Example Group C: High Variability App20.

The quality of the source code of software applications is a factor that directly influences energy consumption (Guamán et al., 2023). Therefore, metrics such as Source Lines of Code (SLOC), Cyclomatic Complexity, Duplicated Lines and Code Smells have been extracted from each of the applications classified in the 3 groups using SonarQube as a static analysis tool. The objective of analyzing the metrics extracted with SonarQube is to determine whether these 3 groups could be aligned with more than just energy consumption and code quality. As a result, applications in *Group a* have a high variability in the quality of their code. Applications belonging to *groups a and c* show optimal quality parameters, while those in *Group b* present inefficient quality values.

Finally, from the analysis of the 3 groups from the point of view of energy consumption, it is possible to conclude that *Group a* contains the applications that consume the lowest energy, while *Group c* characterizes those applications that consume the most, which shows that *applications with a balanced consumption pattern are generally those that consume the lowest energy, while those that show high variability fluctuations belong to the set of applications that consume the most.*

4.2 Measurements in Context 2

The applications to be monitored are obtained from the *GitHub* repository. In this case, applications that are already “*dockerized*” have been selected, that is, those that contain the file called *Dockerfile* to deploy their container. To run an application from a repository on *GitHub*, it is necessary to clone the repository into your local machine and navigate to the root of the project using the terminal. It should be used *Docker* and *Docker Compose* to create and run the containers. **Measurement:** To extract the metrics and energy consumption values of the 4 applications within this context, 4 experiments are performed for each application where different functionalities are executed or the application is stressed for each container that contains the application. In all cases, the functionalities used are CRUD operations.

Table 2: Experimental scenarios.

AppID	Application	Monitored application description	URL Github
App1	Noteapp (Web Application)	Application that uses Flask framework, MariaDB for the database and deployed in three different docker containers.	https://github.com/mosudi/noteapp_docker
	Domain: Education	Functionality: Dashboard and CRUD operations of notes.	
App2	Flask-crud-docker (Web Service)	Application that uses Flask framework, SQLAlchemy for the database and deployed in a single docker container.	https://github.com/mengpellee/flask-crud-docker
	Domain: Education	Functionality: CRUD operations on titles.	
App3	Angular-forum (Web Application)	Application that uses Angular framework, deployed in a single container.	https://github.com/hanzahamidi/angular-forum
	Domain: Education	Functionality: User authentication, user and articles CRUD operations.	
App4	Utpl-thesis (Backend - Microservices)	Microservices that use NestJS framework, Mongoose for the database and Swagger to document each microservices. Deployed in three different containers, one for each microservice.	https://github.com/diegotony/utpl-thesis-order
	Domain: Education	Functionality: GET, POST, PUT, DELETE functionalities for Client, Order, Payment and Billing microservices.	

The objective of carrying out the 4 experiments for each application is: (i) to evaluate the quality of the power estimates by running manual load tests for all containers, including the deployment and each application and the number of containers within it, and (ii) to analyze the accuracy and stability of the *Power-API* power model by running the different workloads or stresses of the application monitored in our experiments (see Table 2). To monitor and extract data from each experiment, each application is evaluated using the *functionality* operation, keeping the same times and workloads. The measurement values are calculated as averages of different execution times for our

containers to be monitored with the metrics of each container configured in our formula. In this case, it is used the *CPU consumption*, since the estimation of the DRAM component (RAM memory) is not yet enabled for use. The results of the measurements obtained in the experiments are presented as follows.

First Experiment: The energy consumption of the “*NoteApp application*” that was run at various time intervals In this experiment, the functionalities contained in the application are executed: add, list, edit and delete notes, the CPU power value measured in Watts/second has been extracted on average, maximum, minimum and the time interval of the functionality’s execution.

Table 3: NoteApp Container Monitoring.

Functionality	MonitoringID	CPU value (average) (W)	CPU value (maximum) (W)	CPU value (minimum) (W)	Execution time (seconds)
Add note	1	0.069	0.788	0.025	60
Add note	2	0.126	1.17	0.0419	60
Add note	3	0.087	1.54	0.006	71
Add note	4	0.217	1.77	0.072	51
List and edit note	1	0.143	0.763	0.038	60
List and edit note	2	0.194	1.66	0.021	60
List and edit note	3	0.074	0.962	0.006	70
List and edit note	4	0.283	2.24	0.019	52
List and delete note	1	0.161	0.921	0.037	30
List and delete note	2	0.132	0.391	0.0471	30
List and delete note	3	0.109	0.855	0.007	27
List and delete note	4	0.215	1.94	0.025	22

Second Experiment: The results obtained by monitoring the “*FlaskApp*” are presented in the table, where the energy consumption is shown at different time intervals in which the application is running and the average, maximum and minimum energy consumption values were saved.

Table 4: FlaskApp Container Monitoring.

Functionality	MonitoringID	CPU value (average) (W)	CPU value (maximum) (W)	CPU value (minimum) (W)	Execution time (seconds)
Add title	1	0.715	3.45	0.34	41
Add title	2	0.254	1	0.141	28
Add title	3	0.445	1.81	0.161	62
Add title	4	0.445	1.81	0.161	62
Edit title	1	0.864	6.21	0.283	30
Edit title	2	0.308	1.34	0.15	29
Edit title	3	0.536	2.48	0.184	58
Edit title	4	0.536	2.48	0.184	58
Delete title	1	1.833	4.74	0.082	11
Delete title	2	0.443	1.36	0.155	13
Delete title	3	0.819	2.45	0.214	27
Delete title	4	0.819	2.45	0.214	27

Third Experiment: The results obtained from monitoring the “*Angular-forum*” application are presented in Table 5. During each time interval, several iterations are performed for the various pages of the application’s Frontend. For each functionality, the average, maximum and minimum values of energy consumption are extracted during the period in which the application was used.

Table 5: Angular-forum Container Monitoring.

Functionality	MonitoringID	CPU value (average) (W)	CPU value (maximum) (W)	CPU value (minimum) (W)	Execution time (seconds)
Login	1	0.016	0.143	0.002	15
Edit article	1	0.007	0.12	0.001	20
Create article	1	0.026	0.183	0.002	17
Update user	1	0.063	0.271	0.002	17
Home	1	0.014	0.171	0.002	17

Fourth Experiment: The “Utpl-Thesis” application has a particularity with regard to the other 3 applications executed in this Context 2, since it is built under the microservices architecture running each microservice (Client, Order, Payment and Billing service) in a container independent of the rest of the application. The results of monitoring are presented in Table 6. Each time interval corresponds to a different functionality within the application, considering each microservice that contributes to the overall structure of the application. This is valid regardless of the number of containers that make up each microservice.

Table 6: Client Microservice monitoring in Containers.

Functionality	Monitoring ID	Container Name	CPU value (average) (W)	CPU value (maximum) (W)	CPU value (minimum) (W)	Execution time (seconds)
Get Client GET HTTP Method	1	utpl-thesis-client_client-micro_1	0.477	4.98	0.032	18
	1	mongo_db_client	0.459	2.49	0.038	18
	1	redis_client	0.207	1.58	0.011	18
Create Client POST HTTP Method	1	utpl-thesis-client_client-micro_1	0.008	1.88	0.0001	68
	1	mongo_db_client	0.225	2.44	0.035	68
	1	redis_client	0.094	1.76	0.011	68
Edit Client PUT HTTP Method	1	utpl-thesis-client_client-micro_1	0.028	2.14	0.0002	28
	1	mongo_db_client	0.694	2.64	0.0394	28
	1	redis_client	0.389	1.82	0.011	28
Delete Client DELETE HTTP Method	1	utpl-thesis-client_client-micro_1	0.0131	1.57	0.0001	54
	1	mongo_db_client	0.392	2.74	0.037	54
	1	redis_client	0.187	1.75	0.011	54

Results and Analysis of Experiments: Based on the experiments for each scenario, the implementation of PowerAPIDocker-Cloud, the use of the HWPC sensor and the SmartWatts formula are validated. The experiments show the behavior of the applications and how the energy consumption of each container affects different scenarios. It is also shown that PowerAPI adapts to evaluate different types of applications and how it responds to variations in workload. The energy consumption analysis of the experiments shows significant variations in CPU usage and execution times for different applications and their specific functionalities. In the *NoteApp Container*, CRUD operations generate various power consumption values (see Table 3). The add note operation generates average CPU consumption values ranging from 0.069 W/s to 0.217 W/s, with maximum peaks reaching up to 2.24 W/s. The execution time for these operations varies between 22 and 71 seconds, which shows notable differences in energy efficiency and processing time required for each operation. The *FlaskApp* exhibits high energy consumption values compared to the other ap-

plications analyzed. The operations for adding, editing and deleting titles have average CPU consumption values ranging from 0.254 W/s to 1.833 W/s, with maximum peaks reaching up to 6.21 W/s. Despite the high energy consumption, execution times are relatively short, ranging from 11 to 62 seconds. This high consumption can be attributed to the application deployment and processing load during CRUD operations, suggesting a higher resource intensity. The client service of the *Utpl-Thesis application*, the energy consumption values vary considerably (see Table 6). In *client service*, average CPU values range from 0.008 W/s to 1.601 W/s, with maximum peaks reaching up to 4.98 W/s. This data indicates that the *microservices architecture of Utpl-Thesis* involves considerable energy consumption, reflecting the complexity and interaction between the different components of the system. It can be concluded that the applications under measurement that have an implementation of CRUD operations and the implementation of services have a significant impact on CPU usage and execution times. In contrast, more complex and processing-intensive applications such as *FlaskApp* and *Utpl-Thesis* show higher power consumption and varied execution times. These findings highlight the importance of optimizing application design and implementation to improve energy efficiency and reduce environmental impact.

5 DISCUSSION

Based on the results presented in Section 4, it is possible to answer the research questions.

RQ1: Is it possible to measure the energy consumption of software applications programmed with different software architectures and programming languages using a containerized PowerAPI environment? Based on the collected data using PowerAPIDocker-Cloud from the measurements of the 2 contexts of the experiment, it can be stated that it is possible to measure the energy consumption of applications that are built with different software architectures and deployed in Docker containers. However, the efficiency and accuracy of these measurements can vary depending on several factors related to: the software architecture of application, the quality and use of good programming practices, the implementation of the containers, and the configuration of the measurement environment. PowerAPIDocker-Cloud is a suitable tool for measuring the energy consumption of applications regardless of the architecture used, since it obtains metrics and data that allow comparing the energy impact generated by the application and monitoring

them to identify areas of optimization and improvement in the design and construction of software and the resource management. Another advantage of using PowerAPIDocker-Cloud is that it can be configured to monitor various applications and services deployed in Docker containers, allowing its use in heterogeneous and dynamic environments. Based on the results obtained, PowerAPIDocker-Cloud is a configurable and parametrizable environment that can be reproduced by organizations or engineers who use multiple technologies and diverse architectures for their applications, although it is important to keep in mind that application measurements will depend on the workload generated by each functionality, the system configuration and other external factors such as code quality, programming languages and technologies.

RQ2: Does the execution time of a functionality influence in the energy consumption or CPU values measured by PowerAPIDocker-Cloud? Based on the data obtained with PowerAPIDocker-Cloud, it is evidenced that the execution times of the functionalities of the applications significantly influences in the energy consumption and CPU value. The execution time of a functionality (i.e. Create, Read, Update or Delete in the case of CRUD systems under analysis) in a software application is directly related to the energy consumption and CPU usage values measured by PowerAPIDocker-Cloud. For example, in the NoteApp WEB application, operations with longer execution times, such as adding notes, can take up to 71 seconds. These operations tend to have higher average and maximum CPU values compared to faster operations such as listing and deleting notes, which take between 22 and 30 seconds. This example shows that power consumption is an important performance measure like processing time, and although power consumption and execution time do not always have a linear relationship, generally, a longer execution time implies a higher power consumption due to the prolonged activity of the processor and other system components. In addition, from the results, it has been observed that running the *add a note* functionality can consume an average CPU value of up to 0.217 W/s, while *deleting a note* has an average consumption of 0.215 W/s. The monitoring results of FlaskApp show a pattern of execution times of less than 10 seconds when the operation involves deleting data. They have lower average and maximum CPU values compared to adding or editing titles, which can take up to 62 seconds. For example, removing a title has an average CPU value of 0.443 W/s, while adding a title can reach 0.715 W/s. This suggests that longer and possibly more complex operations consume more power than those that take less time. As a result, it is

possible to conclude that when applications include operations or methods that require more execution time, they commonly consume more energy due to the longer duration of computational resource usage. The complexity and duration of the operation also have a direct impact on energy consumption. Applications and services with efficient implementations optimized for fast operations tend to have lower power consumption, highlighting the importance of optimizing both code and infrastructure to improve energy efficiency.

RQ3: Does the type of CRUD (Create-Read-Update-Delete) operation of an application influence the energy consumption? What kind of operation (Create-Read-Update-Delete) has the highest energy consumption? In addition, it has been individually analyzed the different kind of CRUD operations, i.e. create, read, update, and delete, to determine to what extent they affect power consumption in a different way. From the results obtained, it is possible to determine that the data entry and editing operations, i.e., create and update, tend to have higher power consumption than the list and delete operations due to factors such as database operations, inclusion of business rules, memory usage, and transactions. When performing database operations, there is a higher input/output load, since when data are inserted or updated, the system must write to disk or the database, which is more energy-intensive than just reading information. In the case of web applications and microservices, before adding or modifying a record, validations (format, uniqueness, referential integrity) are performed, which requires additional processing in CPU and memory. Similarly, in the case of applications that use microservices, data must be transformed from a JSON to a data structure in our example, which implies an extra computational expense. In addition, both web applications and microservices require transactions to ensure consistency, which implies temporary storage and extra processing in the database engine. Furthermore, in the case of applications with microservice architecture, microservices also require coordination processes that also increase the energy consumption. Therefore, it is possible to conclude that the kind of operation and the architecture of the application have a high influence in the power consumption.

6 CONCLUSIONS

This paper presents the containerization of PowerAPI in an environment called PowerAPIDocker-Cloud, which makes it possible to measure and evaluate the energy consumption generated by software applica-

tions. Key concepts, features and functionalities of PowerAPIDocker-Cloud related to the measurement and estimation of energy consumption of software applications in real time were identified, since the HWPC sensor and the Smartwatts formula are configured as part of its architecture, in addition to the MongoDB, influxDB and Grafana services. To validate the architecture proposed by PowerAPIDocker-Cloud, 33 experiments were designed for two different contexts to demonstrate that it is able to measure effectively different MVC applications programmed in different languages and deployed in a different way: desktop, web, web service and microservices. In addition, this paper analyzes in detail the power consumption patterns of the application under measurement, providing relevant findings to software engineers about the IDEs' consumption, the kind of CRUD operation and the execution times that can be applied during software development.

The features and functionalities identified in PowerAPI are numerous; however, they are not yet fully developed in relation to the measurement and estimation of energy consumption in real time. This will allow future research to achieve a deeper understanding of the capabilities of PowerAPI and take advantage of its potential to optimize the energy consumption of microservices. To replicate the PowerAPIDocker-Cloud setup for estimating microservices power consumption, it is suggested to carefully study the information related to the middleware, which provides a solid foundation for understanding the importance and implications of using PowerAPI, which can help make informed decisions about its implementation and utilization.

ACKNOWLEDGEMENTS

This work is partially supported by Universidad Técnica Particular de Loja (Computer Science Department) and the Spanish Ministry of Science and Innovation (MICINN) through "SIoTCom: Sustainability-Aware IoT Systems Driven by Social Communities" (PID2020-118969RB-I00).

REFERENCES

- Agarwal, S., Nath, A., and Chowdhury, D. (2012). Sustainable approaches and good practices in green software engineering. *International Journal of Research and Reviews in Computer Science*, 3(1):1425.
- Bozzelli, P., Gu, Q., and Lago, P. (2013). A systematic literature review on green software metrics. *VU University, Amsterdam*.
- David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. (2010). Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pages 189–194.
- Fieni, G., Acero, D. R., Rust, P., and Rouvoy, R. (2024). Powerapi: A python framework for building software-defined power meters. *Journal of Open Source Software*, 9(98):6670.
- Grant, R. E., Levenhagen, M., Olivier, S. L., DeBonis, D., Pedretti, K., and Laros, J. H. (2016). Overcoming challenges in scalable power monitoring with the power api. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1094–1097. IEEE.
- Guamán, D., Pérez, J., Díaz, P. V., and Canas, N. (2022). Estimating the energy consumption of software components from size, complexity and code smells metrics. In Hong, J., Bures, M., Park, J. W., and Cerný, T., editors, *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022*, pages 1456–1459. ACM.
- Guamán, D., Pérez, J., and Valdiviezo-Díaz, P. (2023). Estimating the energy consumption of model-view-controller applications. *The Journal of Supercomputing*, 79(12):13766–13793.
- Kocak, S. A. (2013). Green software development and design for environmental sustainability. In *11th International Doctoral Symposium on Empirical Software Engineering (IDOESE 2013)*. Baltimore, Maryland, volume 9.
- Liu, K., Pinto, G., and Liu, Y. D. (2015). Data-oriented characterization of application-level energy optimization. In *Fundamental Approaches to Software Engineering: 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 18*, pages 316–331. Springer.
- Mancebo, J., Garcia, F., and Calero, C. (2021). A process for analysing the energy efficiency of software. *Information and Software Technology*, 134:106560.
- Pazowski, P. et al. (2015). Green computing: latest practices and technologies for ict sustainability. In *managing intellectual capital and innovation for sustainable and inclusive society, joint international conference, Bari, Italy*, pages 1853–1860.
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830.
- Rad, B. B., Bhatti, H. J., and Ahmadi, M. (2017). An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228.
- Torvekar, N. and Pravin, S. G. (2019). Microservices and it's applications: An overview. *International Journal of Computer Sciences and Engineering*, 7(4):803–809.